Abstract Presentation Back to Agenda
Next Abstract

Multimedia Macros for Portable Optimized Programs

Juan Carlos Rojas and Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University, Boston

Abstract

Multimedia architectures can speed-up applications significantly when programmed manually. Optimized programs have been non-portable up to now, because of differences in instruction sets, register lengths, alignment requirements and programming styles. We solve all these problems by using a library of C pre-processor macros called MMM. We implemented three examples from video compression in MMM, and automatically translated them into optimized code for four distinct multimedia processors. Their performance is comparable, and in several cases better, than equivalent examples optimized by the processor vendors.

Problem

Multimedia computing has been one of the greatest challenges in computer engineering for the last decade. Computer designers have been challenged to come up with solutions capable of processing the enormous amounts of data required by multimedia applications. The solutions came in the form of multimedia processors, and multimedia extensions to general-purpose processors. Some well-known examples are AltiVec, MMX and its successors – SSE and SSE2, and TriMedia processors.

All multimedia architectures follow the same basic approach: they partition the registers into sections that represent multiple data elements, and operate on all the sections in parallel. In addition, complex instructions have been added to speed-up specific tasks found in multimedia applications. For example, some architectures include a Sum of Absolute Differences instruction, or a Multiply-High instruction (multiply and pack the most-significant part of the product).

Our experiments and other published results show that multimedia architectures can speed-up applications by factors of up to 15, but manual optimization is required in order to take full advantage of the complex instructions available. Manual optimization is very time consuming, and up to now has resulted in non-portable programs. This is in part because different multimedia architectures have different register lengths, different programming styles, different alignment requirements, and they support different partitioned instructions.

Solution: MMM

We solved the problem by creating MMM: a library of target-independent C pre-processor macros that implements a common set of parallel operations available or efficiently emulated on a given set of target architectures. MMM provides a unique interface to architectures with different register lengths and instruction sets. Long data vectors are simulated by using several small vectors, and operations of long vectors are emulated as a sequence of operations on short vectors. Similarly, vector operations that are missing on a given target are emulated using a sequence of simple vector operations, when it is efficient to do so. The same concept can be used to resolve different alignment requirements. Some architectures require that vector loads and stores are done at aligned addresses. If an unaligned load is required, one must load two aligned vectors and compose the desired result from them. All this can be encapsulated inside MMM load macros, and thus provide the programmer with a general unaligned load virtual instruction.

The table below shows simplified MMM macro definitions in different targets of the Sum of Absolute Differences of two 128-bit vectors partitioned into 8-bit sections. Two partial sums are returned in a vector.

SSE2 (128-bit registers)	AltiVec (128-bit registers)

`#define SAD_U8x16(a,b,c) \`
`  a = _mm_sad_epu8(b,c);`
	`#define SAD_U8x16(a,b,c) \`
	`  a = vec_sum2s(vec_sum4s(vec_sub(vec_max(b,c), vec_min(b,c)));`

MMX+SSE (64-bit registers)	**TriMedia (32-bit registers)**

`#define SAD_U8x16(a,b,c)            \`
`  a##_0 = _m_psadbw(b##_0, c##_0); \`
`  a##_1 = _m_psadbw(b##_1, c##_1);`
	`#define SAD_U8x16(a,b,c)          \`
	`  a##_0  = UME8UU(b##_0, c##_0); \`
	`  a##_0 += UME8UU(b##_1, c##_1); \`
	`  a##_1  = UME8UU(b##_2, c##_2); \`
	`  a##_1 += UME8UU(b##_3, c##_3);`

Through emulation, MMM implements a large common virtual instruction set for several target architectures. By using MMM, it is possible to write multimedia applications that are portable among different multimedia processors, and take advantage of the complex partitioned operations available on them.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **20 AUG 2004**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **Multimedia Macros for Portable Optimized Programs**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Department of Electrical and Computer Engineering Northeastern University, Boston**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES

See also ADM001694, HPEC-6-Vol 1 ESC-TR-2003-081; High Performance Embedded Computing (HPEC) Workshop (7th)., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**33**	

Abstract Presentation

Back to Agenda

Next Abstract

Multimedia Macros for Portable Optimized Programs

Juan Carlos Rojas
Miriam Leeser
Northeastern University
Boston, MA

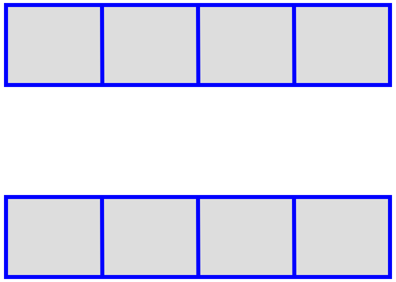
Programming Multimedia Architectures

- Many different Multimedia Architectures
  - TriMedia®, AltiVec™, MMX™, SSE, SSE2
  - Different complex parallel instructions
  - Different register lengths, data types, ...
- Goals for programming
  - Portability
    - Write application once
    - Translate to any architecture

  - High Performance
    - Make best use of each architecture

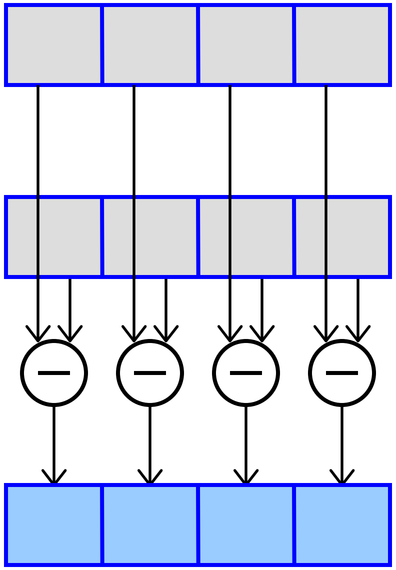
Multimedia Architectures

- Partitioned registers



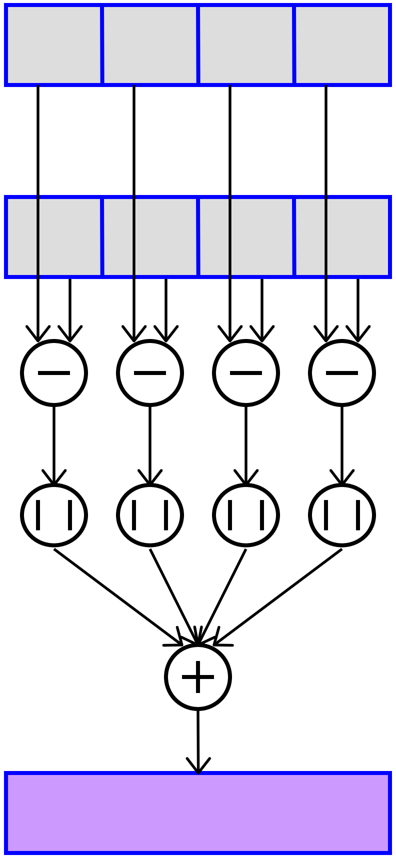
Multimedia Architectures

- Partitioned registers
- Parallel operations



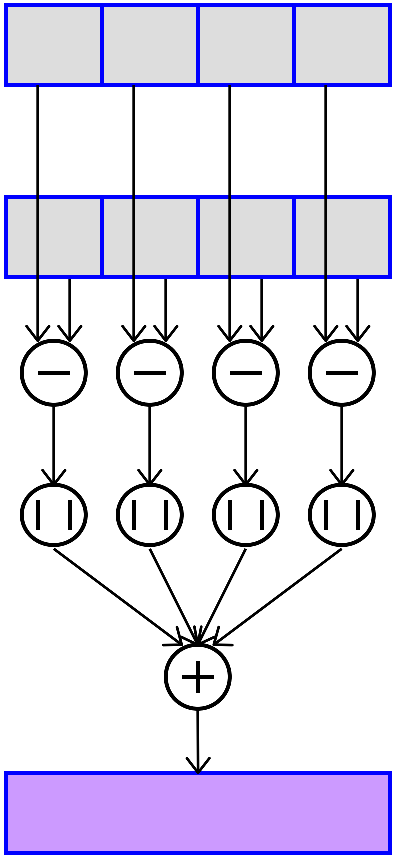
Multimedia Architectures

- Partitioned registers
- Parallel operations
- Complex instructions

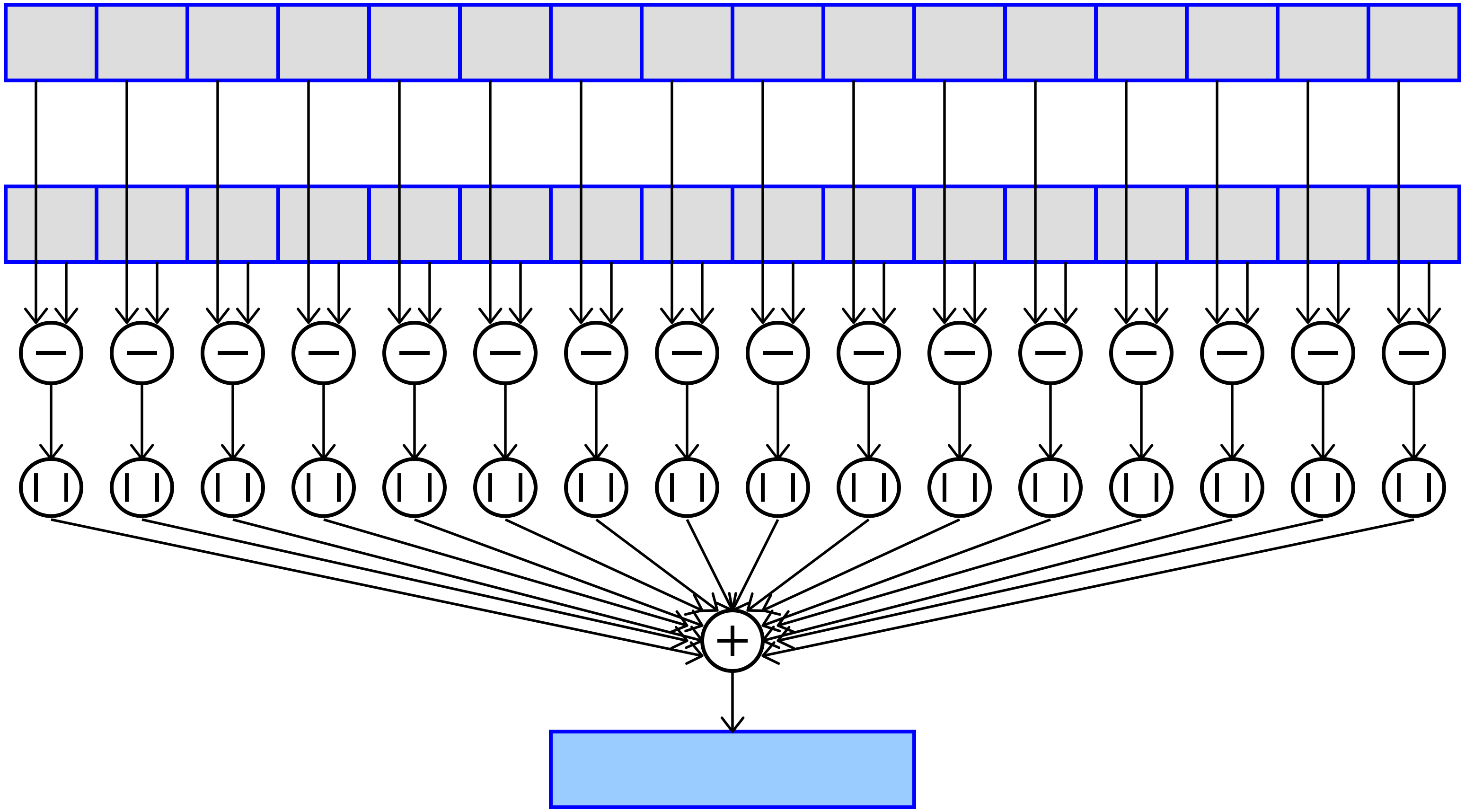


Multimedia Architectures

  - Partitioned registers
  - Parallel operations
  - Complex instructions
- Architectures differ in
  - Register lengths
  - Instruction sets
  - Alignment requirements
  - Programming styles



Example: Vector SAD

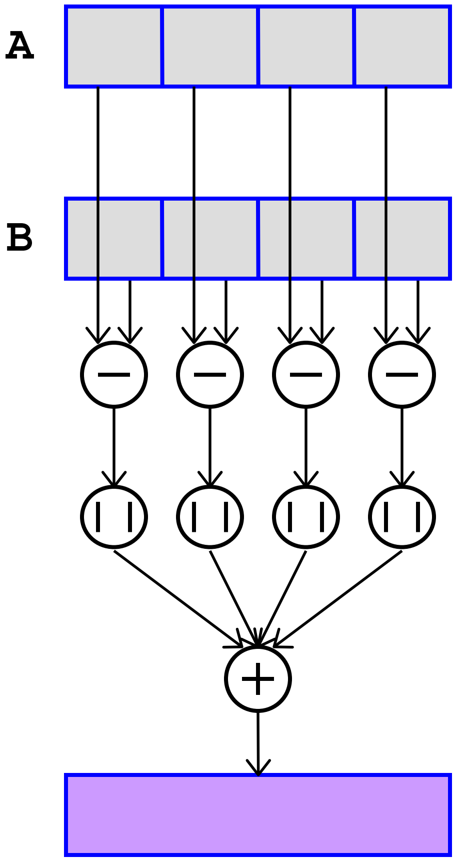


Vector SAD: Scalar Implementation

```
uint8 *a, *b;
int diff, sad;

sad = 0;
for (i=0; i<16; i++)
{
    diff = a[i] – b[i];
    sad += diff > 0 ? diff : -diff;
}
```

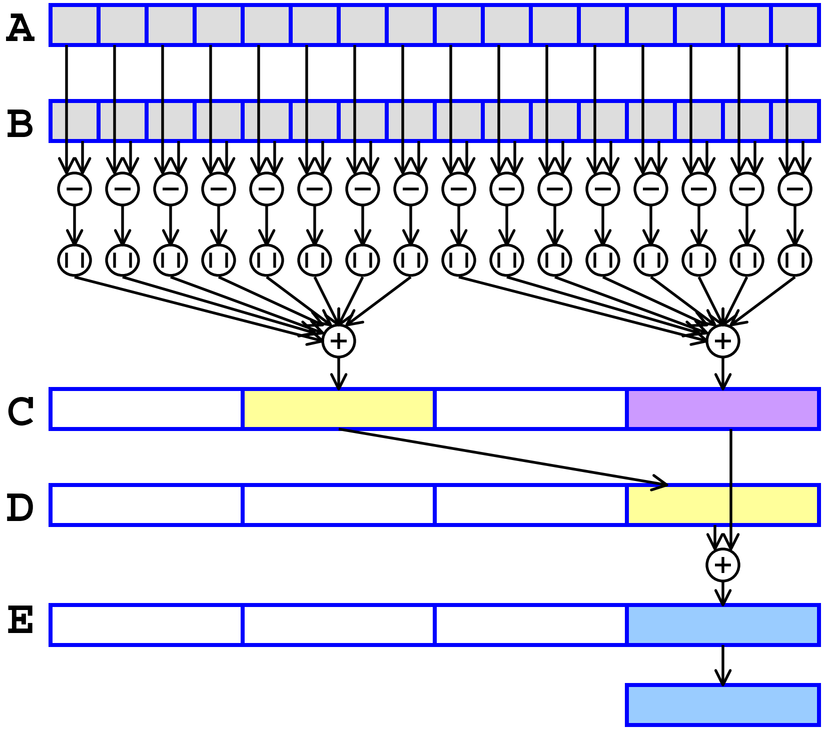
Vector SAD: Optimized for TriMedia®

```
uint8 *a, *b;
int A, B, sad;

A = *((int *) a);
B = *((int *) b);
sad = UME8UU(A, B);
A = *((int *)(a+4));
B = *((int *)(b+4));
sad += UME8UU(A, B);
A = *((int *)(a+8));
B = *((int *)(b+8));
sad += UME8UU(A, B);
A = *((int *)(a+12));
B = *((int *)(b+12));
sad += UME8UU(A, B);
```



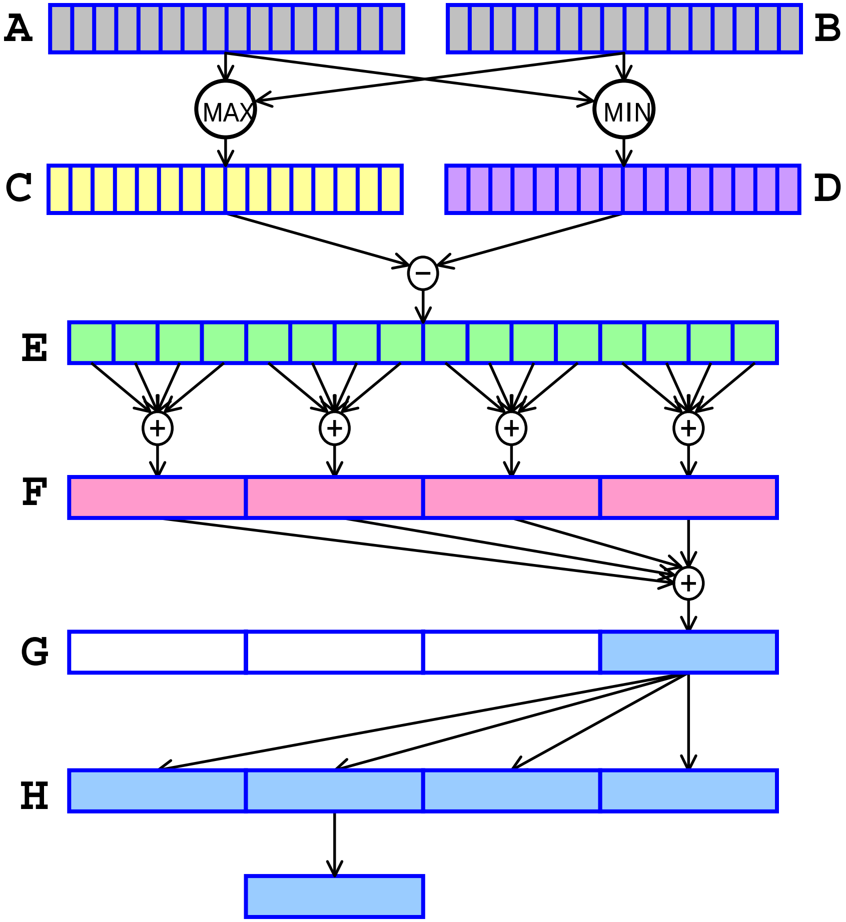
Vector SAD: Optimized for SSE2

```
uint8 *a, *b;
__m128i A, B, C, D, E;
int sad;

A = _mm_load_si128(
        (__m128i *) a);
B = _mm_load_si128(
        (__m128i *) b);
C = _mm_sad_epu8(A, B);

D = _mm_srli_si128(C, 8);
E = _mm_add_epi32(C, D);
sad = mm_cvtsi128_si32(E);
```



Vector SAD: Optimized for AltiVec™

```
uint8 *a, *b;
vector uint8 A, B, C, D;
vector uint8 E, F, G, H;
int sad;

A = vec_ld((vector uint8 *) a);
B = vec_ld((vector uint8 *) b);
C = vec_min(A, B);
D = vec_max(A, B);
E = vec_sub(C, D);

F = vec_sum4s(E);
G = vec_sums(F);

H = vec_splat(G, 3);
vec_ste(H, &sad);
```

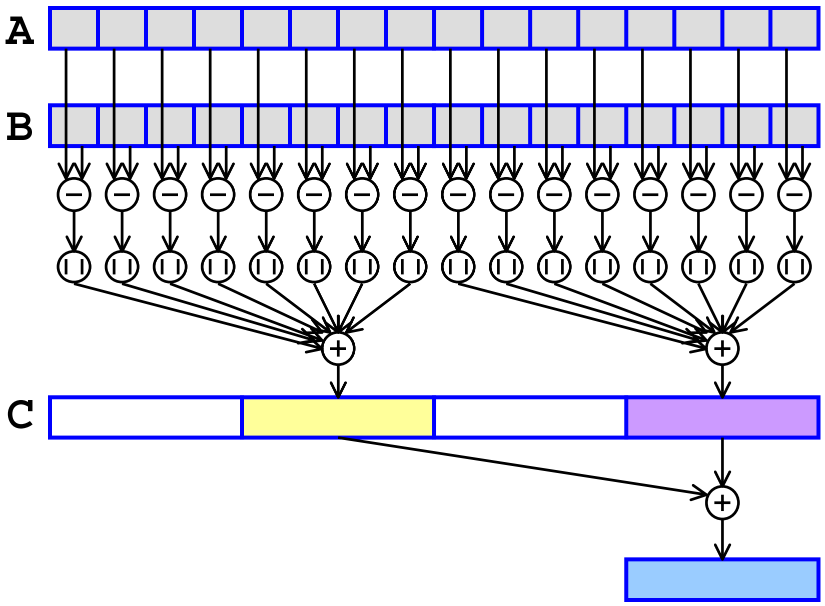


Solution: MMM

- MMM: MultiMedia Macros
- Instruction-level macro library
- Common virtual instruction set
- Emulation of long registers
- Emulation of instructions

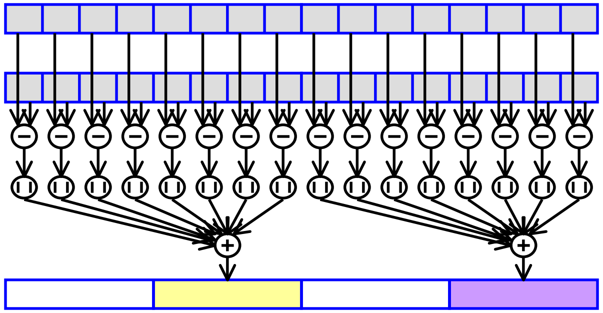
Vector SAD: MMM Version

```
uint8 *a, *b;
DECLARE_U8x16(A)
DECLARE_U8x16(B)
DECLARE_U32x4(C)
int sad;

LOAD_A_U8x16(A, a)
LOAD_A_U8x16(B, b)
SAD2_U8x16(C, A, B)
SUM2_U32x4(sad, C)
```

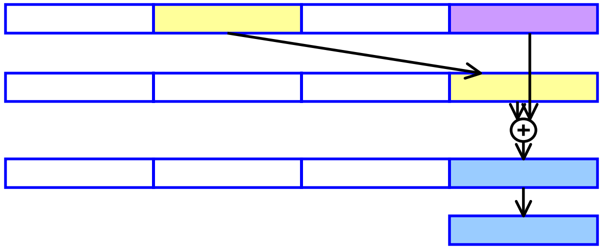


MMM definitions for SSE2

```
#define DECLARE_U8x16(var) \
    __m128i var;

#define LOAD_A_U8x16(var, ptr) \
    var = _mm_load_si128((__m128i *) (ptr));


#define SAD2_U8x16(dst, src1, src2) \
    dst = _mm_sad_epu8(src1, src2);
```

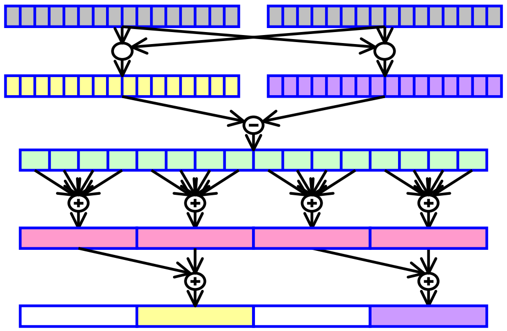


```
#define SUM2_U32x4(dst, src) \
    dst = _mm_cvtsi128_si32( \
          _mm_add_epi32(src, \
          _mm_srli_si128(src, 8)));
```

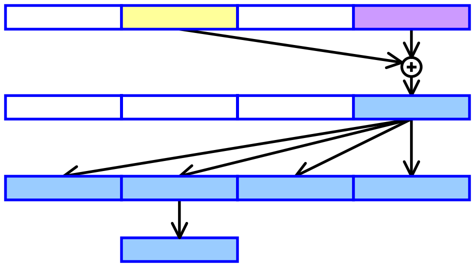


MMM definitions for AltiVec™

```
#define DECLARE_U8x16(var) \
    vector UINT8 var;

#define LOAD_A_U8x16(var, ptr) \
    var = vec_ld((vector UINT8 *)(ptr));


#define SAD2_U8x16(dst, src1, src2)    \
    dst = vec_sum2s(vec_sum4s(         \
          vec_sub(vec_max(src1, src2), \
                  vec_min(src1, src2))));
```

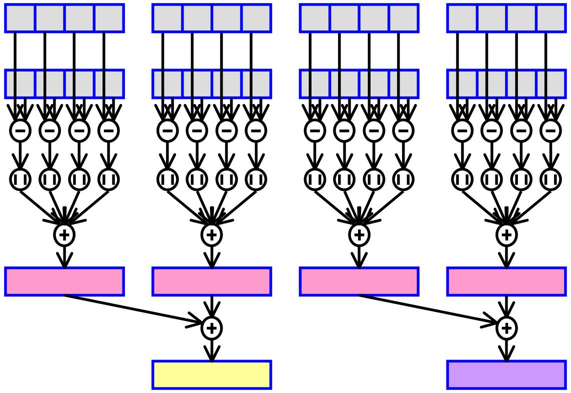


```
#define SUM2_U32x4(dst, src) \
    vec_ste(vec_splat(       \
            vec_sums(src), 3), &dst);
```



MMM definitions for TriMedia®

```
#define DECLARE_U8x16(var) \
    unsigned int var##_0;  \
    unsigned int var##_1;  \
    unsigned int var##_2;  \
    unsigned int var##_3;

#define LOAD_A_U8x16(var, ptr)     \
    var##_0 = *((int *) (ptr));    \
    var##_1 = *(((int *)(ptr))+1); \
    var##_2 = *(((int *)(ptr))+2); \
    var##_3 = *(((int *)(ptr))+3);

#define SAD2_U8x16(dst, src1, src2)       \
    dst##_0 = UME8UU(src1##_0, src2##_0)+ \
              UME8UU(src1##_1, src2##_1); \
    dst##_2 = UME8UU(src1##_2, src2##_2)+ \
              UME8UU(src1##_3, src2##_3);

#define SUM2_U32x4(dst, src) \
    dst = src##_0 + src##_2;
```



Other Approaches

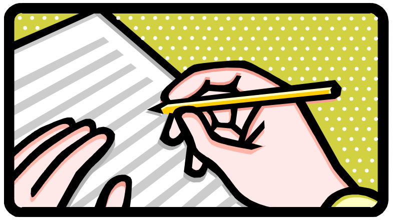
- Parallelizing compilers
- Optimized kernel libraries
  - BLAS, Intel® IPP, VSIPL
- Data-parallel languages
  - Fortran 90, SWARC, Vector Pascal
  - C++ SIMD classes
- Automatic code generators
  - SPIRAL, FFTW, ATLAS
- None of these approaches achieves performance *and* flexibility of MMM
  - MMM makes use of complex instructions in each ISA

MMM Advantages

- General solution

- Complex applications

- Complex partitioned instructions
- Hand-coded performance



Our Approach

- Study representative architectures:
  - TriMedia® TM1300 – 32-bit registers
  - MMX$^{\text{TM}}$+SSE – 64-bit integer registers
  - SSE2 – 128-bit integer registers
  - AltiVec$^{\text{TM}}$ – 128-bit registers
- Define common virtual instruction set
- Implement MMM libraries for each target
- Implement portable applications in MMM
- Measure performance, compare to reference implementations

Common Virtual Instruction Set

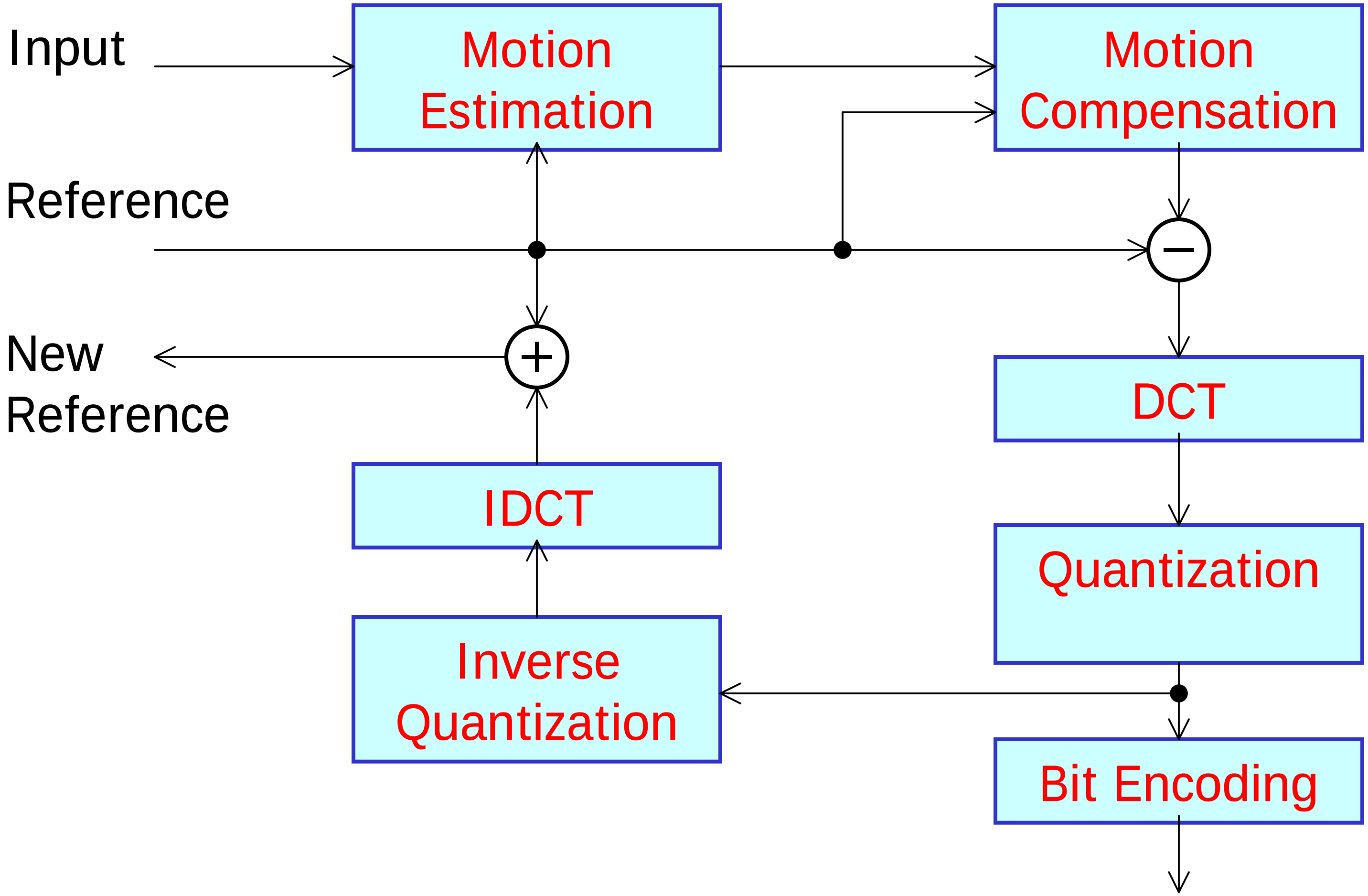
Vector types	**I8x16 U8x16 I16x8 U16x8** **I32x4 U32x4 F32x4**
Vector declarations	**DECLARE DECLARE_CONST**
Set	**SET SET1 CLEAR COPY**
Load and store	▪ Aligned ▪ Unaligned
Rearrangement	**INTERLEAVE BROADCAST PERMUTE**
Type conversion	**PACK EXTEND CVT** ▪ Integer <-> float ▪ Vector <-> scalar

Common Virtual Instruction Set

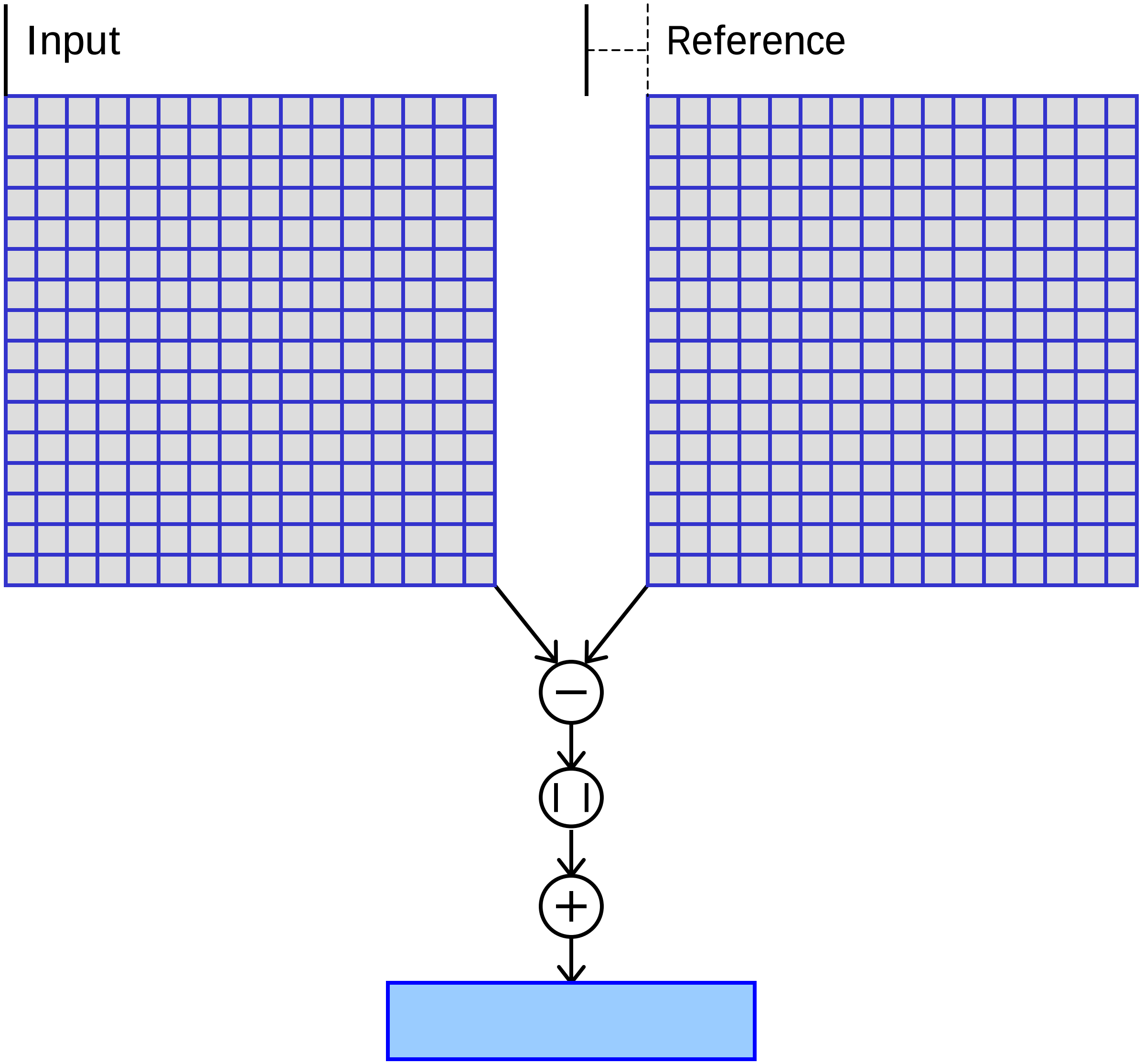
Shift	SLL SRL SRA ROL SLL_I SRL_I SRA_I ROL_I
Bit-wise logic	AND ANDN OR XOR SEL
Comparison	CMP_EQ CMP_LT CMP_GT
Float arithmetic	ADD SUB MULT MULT_ADD DIV MIN MAX SQRT
Integer arithmetic	ADD SUB AVG MIN MAX MULT_H MULT_L MULT_ADDPAIRS SAD2 SUM2 Handling of overflow: ▪ Modulo ▪ Saturation ▪ Unspecified

Example Programs

MPEG2 Video Encoder



16x16 Block L_1-Distance



16x16 Block L_1-Distance

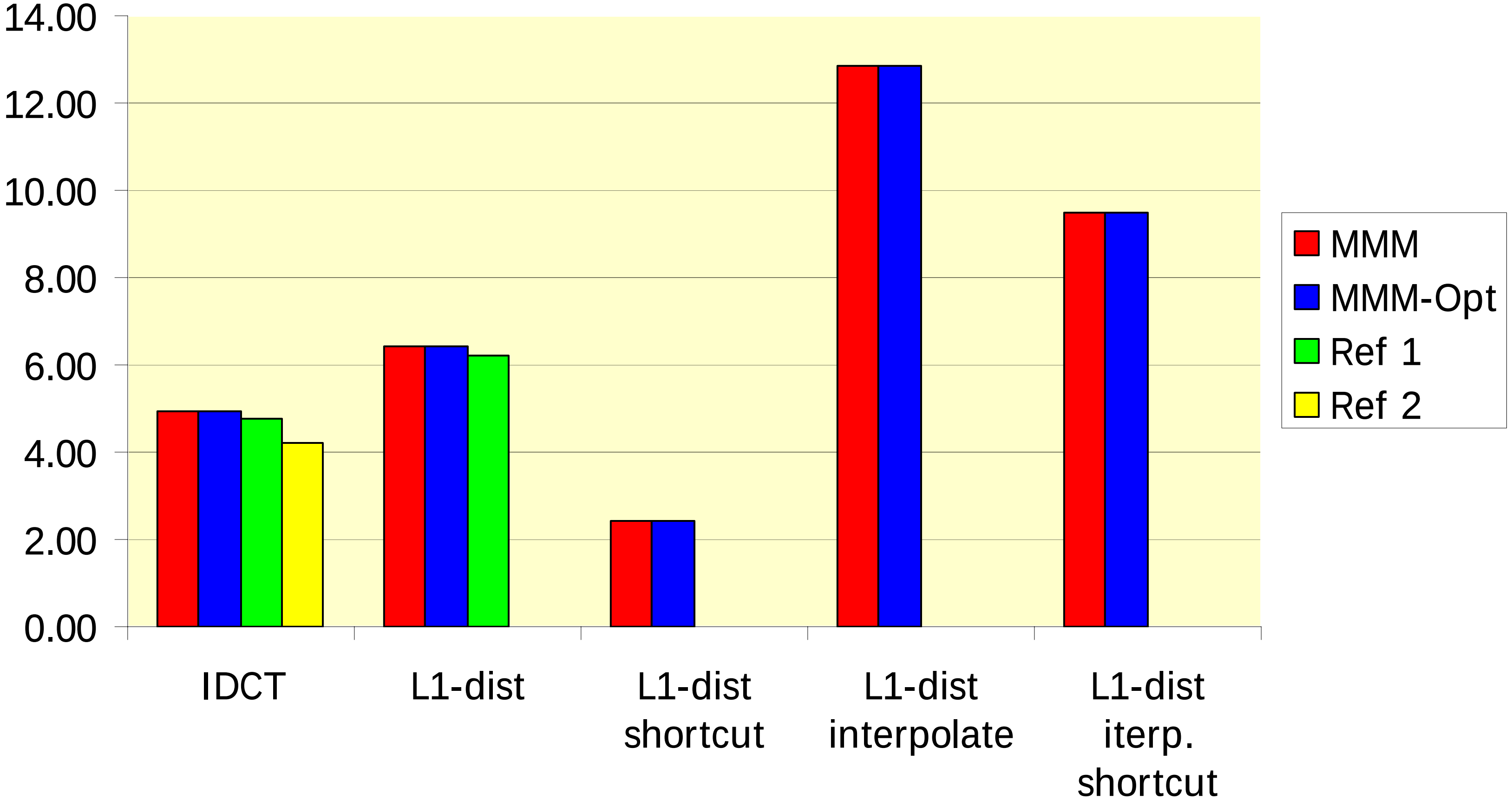
```
DECLARE_U8x16(R1)
DECLARE_U8x16(I)
DECLARE_U32x4(Sad)
UINT32 Sum;
CLEAR_U32x4(Sad)
PREPARE_LOAD_ALIGNMENT(1, pRef)
SAD_ROW(Sad, pRef + 0*RowPitch, pIn + 0*RowPitch, 1)
:
SAD_ROW(Sad, pRef +15*RowPitch, pIn +15*RowPitch, 1)
SUM2_U32x4(Sum, Sad)

#define SAD_ROW(dst, pRef, pIn, index)       \
    LOAD_U_U8x16(R1, pRef, index)            \
    LOAD_A_U8x16(I, pIn)                     \
    SAD2_ADD_M_U8x16(dst, R1, I, dst)
```

Example Reference Versions

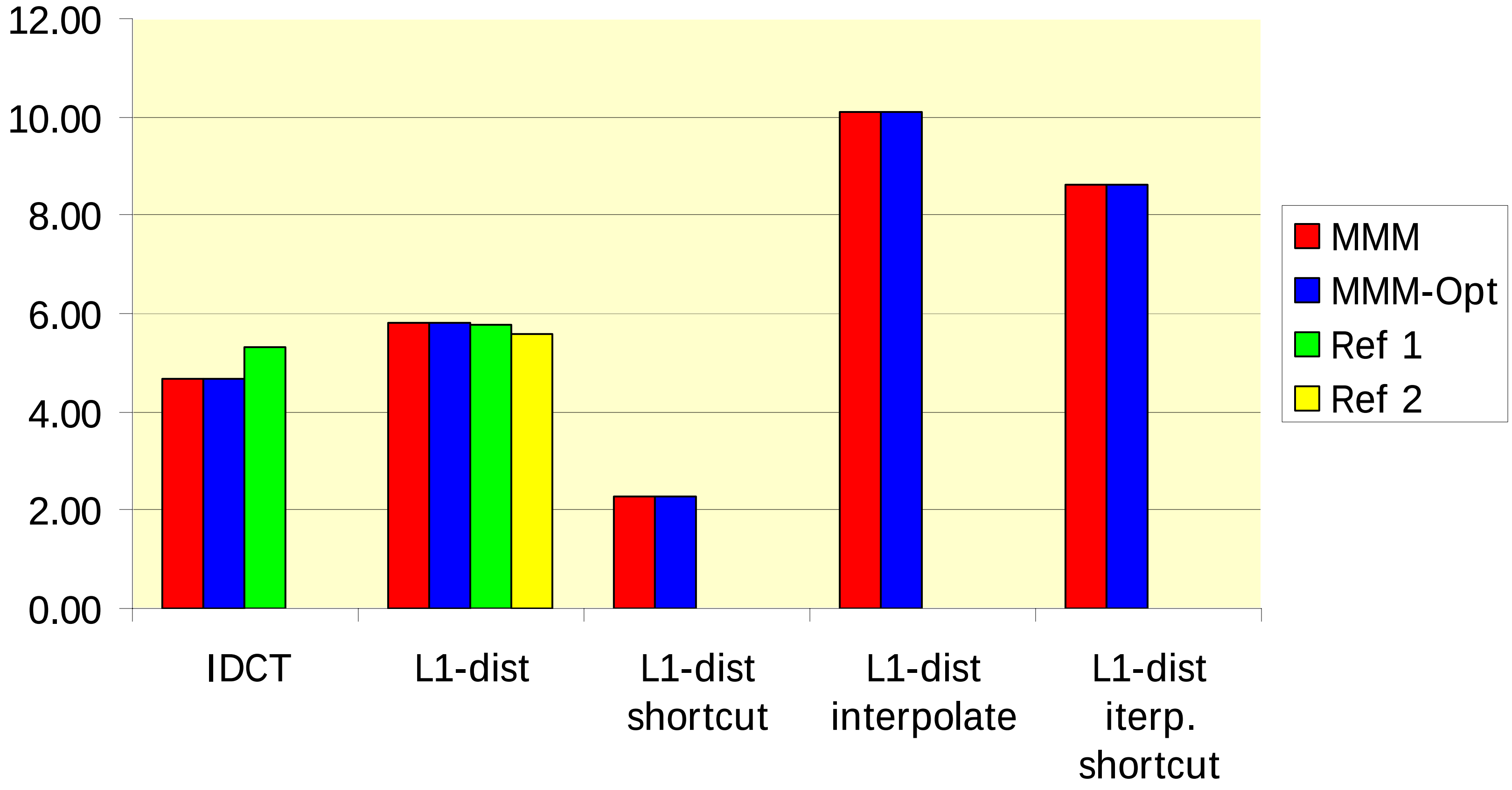
	IDCT	L_1-Distance
TriMedia®	Case study	
MMX™+SSE	Assembly	Assembly C + intrinsics
SSE2	Assembly C++ vector classes	C + intrinsics
AltiVec™	C + intrinsics	C + intrinsics

SSE2 Speedups

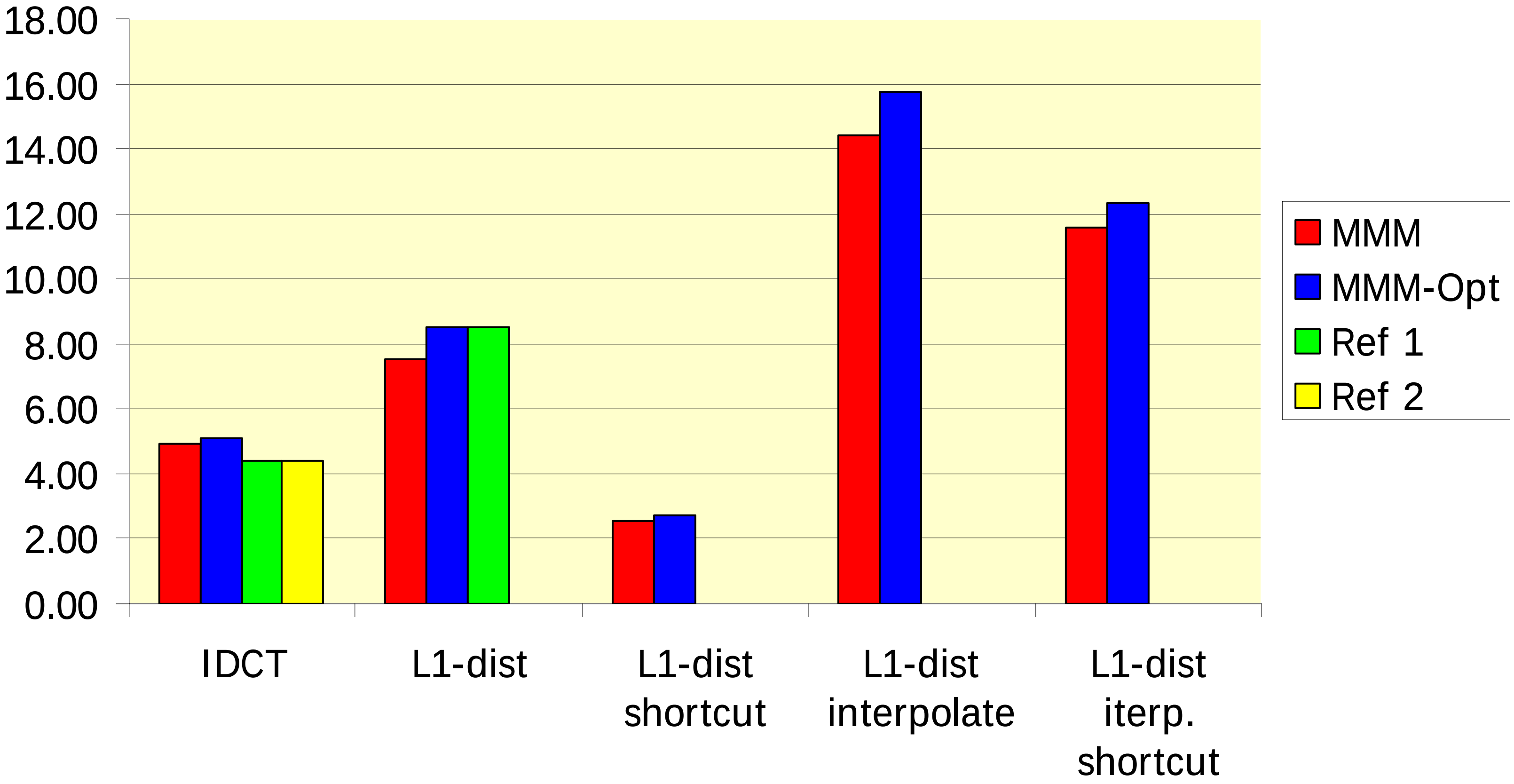
$$Speedup = \frac{Time_{Scalar}}{Time_{Optimized}}$$



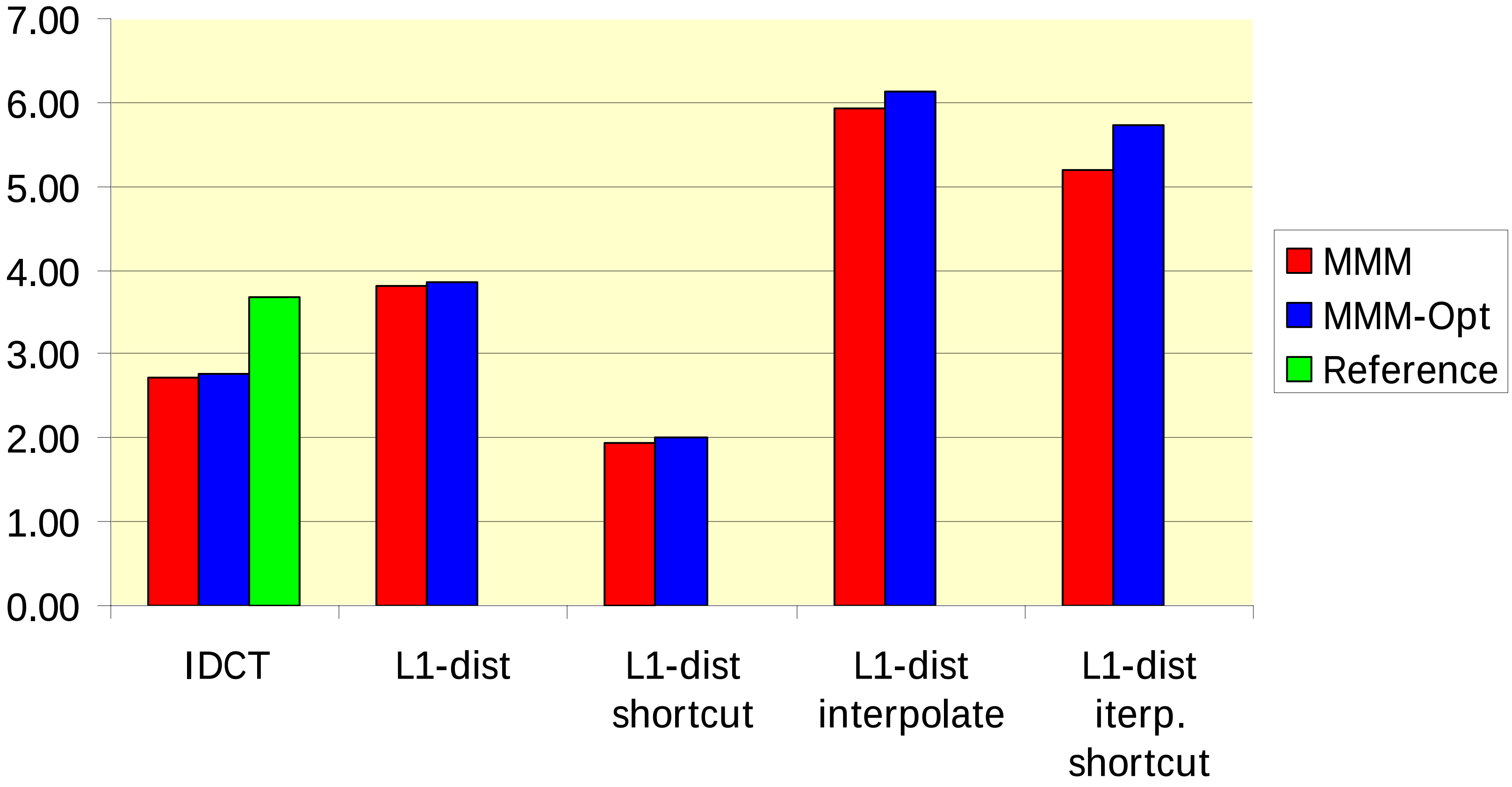
MMX^{TM}+SSE Speedups



AltiVec® Speedups



TriMedia® Speedups



Conclusions and Future Work

- MMM = Portable + Optimized
- Diverse architectures
- Complex examples, complex instructions
- Hand-coded performance
    - Within 12% of best
- Solution can be applied to other ISAs
    - SIMD & DSP
- Future Work:
    - Address ease of programming issues
        - MMC: Multimedia C

Vector SAD: MMC Version

```
uint8 *a, *b;
u8x16 A, B;
u32x4 C;
int sad;

A = *a;
B = *b;
C = SAD2(A, B);
sad = SUM2(C);
```

Abstract Presentation Back to Agenda Next Abstract

Other Approaches

Parallelizing compilers can generate some multimedia instructions from scalar code, but not the most complex ones. The problem is that it is not easy to express these complex parallel instructions in C. One can also write parallel programs explicitly using a data-parallel language, but this still does not solve the problem of expressing complex parallel instructions. Not even languages specifically designed for multimedia [1, 2] can express complex operations like Sum of Absolute Differences, or Multiply-High.

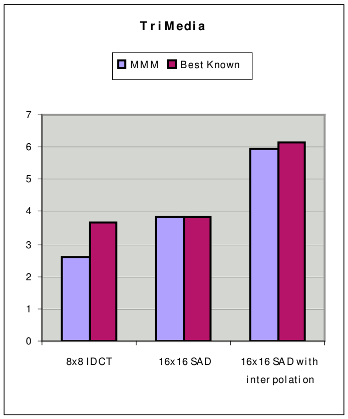
Multimedia code can also be generated from abstract descriptions, like in SPIRAL [4]. This approach is complementary to MMM: the code generator can experiment with different algorithm designs, and prototype them using MMM. Another possibility is to write multimedia applications based on optimized libraries that conform to a standardized API, like VSIPL [3]. This is a good approach for certain classes of applications, but not as flexible as MMM.

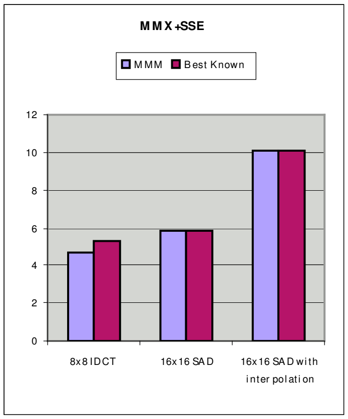
Experiments

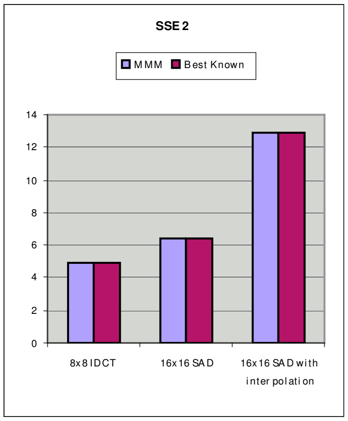
We implemented an MMM library for four distinct multimedia architectures: AltiVec, MMX+SSE, SSE2, and TriMedia TM1300. These architectures are very diverse. Their register lengths vary from 32 to 128 bits, they have distinct instruction sets, alignment requirements and programming styles.

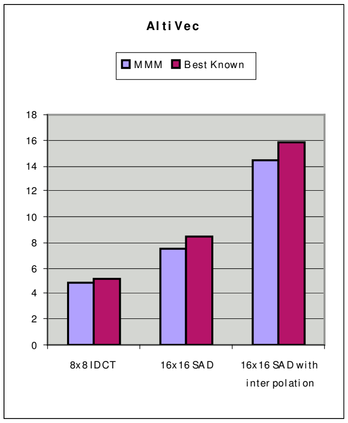
Then we implemented three example programs on MMM used in video compression: 8x8 Inverse Discrete Cosine Transform (IDCT), 16x16 Sum of Absolute Differences (SAD), and 16x16 SAD with horizontal and vertical interpolation. Through MMM libraries, the same programs were automatically converted into optimized code for all four target platforms.

We measured the execution time and instruction count of the MMM programs on all four targets, and compared them with equivalent programs hand-optimized by each processor vendor. In some cases our programs out-performed the vendor examples, so we attempted to further optimize our programs for each target using non-portable instructions, to serve as references. We compared all the optimized programs with reference scalar implementations, and computed the speedup and the reduction in instruction counts. The following charts show the speedup of the portable MMM versions compared to the best known optimized versions for each target:


TriMedia


MMX+SSE


SSE2


AltiVec

The portable MMM programs obtained speedups that are comparable to the best known hand-optimized versions for each target. In some cases, they provide the best known performance. The portable programs written in MMM are indeed optimized for several architectures at the same time.

Conclusions

It is possible to write portable-optimized multimedia programs using MMM. These programs are portable among a diverse group of architectures that have different register lengths, instruction sets, alignment requirement and programming styles. The performance of portable MMM programs is comparable to the best known implementations for each target.

References

[1] Paul Cockshott. *Vector Pascal, an Array Language*. Jan. 2002, http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/vp-ver2.pdf (current May 2003).
[2] Randall J. Fisher, and Henry G. Dietz. “Compiling for SIMD Within a Register,” *Lecture Notes in Computer Science*, vol. 1656, Springer, Berlin, 1998, pp. 292-304.
[3] David Shwartz, et al. *VSIPL 1.01 API,* http://www.vsipl.org/CD/vsiplv1p01_final1.pdf (current May 2003).
[4] Franz Franchetti, and Markus Püschel. “A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms,” *Proceedings International Parallel and Distributed Processing Symposium (IPDPS),* 2002.